

# K Prototype

## De la FSL

This page describes the work-in-progress on defining a K interpreter. K-Maude is an implementation of K on top of Maude language. It fully extends Maude, while adding specific K constructs to facilitate design of programming and domain specific languages.

K-Maude intended audience are language designers who use the K technique.

## Contents



- 1 Obtaining K-Maude
- 2 Basic usage
- 3 More advanced usage
  - 3.1 Using Full Maude
    - 3.1.1 Syntax
    - 3.1.2 Configuration
    - 3.1.3 Semantics
    - 3.1.4 Putting them all together
    - 3.1.5 Entire definition in one module, and a more complex language -- KCHALLENGE
  - 3.2 Using Core Maude
    - 3.2.1 Syntax
    - 3.2.2 Configuration
    - 3.2.3 Semantics
    - 3.2.4 Scripts automating the process
  - 3.3 Generating Latex

## Obtaining K-Maude

The most recent version of K can be accessed through svn at:

```
svn checkout https://subversion.cs.uiuc.edu/svn/FSL/projects/k-meta-defs/
```

WARNING: the current definition is not always stable and not all examples might work with it. Current working examples are **imp?**, **KernelC?**, and **SILF** (only untyped). This version is recommended if you want to take advantage of the latest improvements, but be prepared to encounter problems while using it. Checkout is currently restricted to FSL members. However, if you want to experiment with it, please let us know (mailto:fsl@cs.uiuc.edu) .

A (relatively) stable and (relatively) recent version can be downloaded here:  K-Maude.zip   
(http://fsl.cs.uiuc.edu/index.php/Image:K-Maude.zip) .

## Basic usage

K-Maude comes with a suite of examples defining various programming languages and type checkers/inferencers.

To access them, you must have maude installed and you can generally run any of them by loading the

```
<prefix>-test.maude
```

from the directory containing the **<prefix>-semantics.maude** file. This should run the interpreter on the example programs located in the "**<prefix>-programs.maude**" file. The simplest usage scenario is that the user is satisfied with existing definitions and only wants to try new programs on them. In that case one only modifies the "**<prefix>-programs.maude**", and the corresponding "**<prefix>-test.maude**" from the same directory, then reloads in maude **<prefix>-test.maude** from the directory containing **<prefix>-semantics.maude** to see the outcomes of rerunning the examples.

## More advanced usage

The current K-Maude prototype comes in two flavors. One of them, more stable, but harder to use, is directly running in Maude, in which specifications are concrete Maude modules. The specific K notation is given in this setting by using the metadata attribute (for syntax constructs) and membership axioms (for K equations and rules). The other, easier to use, but less stable, is running on top of Full Maude, extending Full Maude's syntax with the K-specific operations attributes and with keywords introducing K equations and K rules, and adding new syntax for modules.

In general, the user needs to provide three modules for each language definition: one defining the syntax, one defining the configuration of the running state, and one defining the rules for executing the program. The syntax module contains the syntax of the language defined using usual Maude sorts and operations, containing annotations specifying desired order of evaluation, or renaming constructs into semantic ones. It might also contain equations for desugaring auxiliary constructs.

## Using Full Maude

We will exemplify the specifics of the Full-Maude-based implementation using the KERNELC language definition. For each of the three modules, syntax, configuration, and semantics, we will first present the module as it is in the example included with the distribution, then discuss the features which are specific to K, assuming the user familiar with Maude constructs.

### Syntax

```
(k syntax for KERNELC is including GENERIC-EXP-K-SYNTAX + STRING-K-SYNTAX .
sorts Stmt StmtList Pgm .
subsort Stmt < StmtList .
op #include<stdio.h>`#include<stdlib.h>`void`main`(void)`{`_`} : StmtList -> Pgm [renameTo _ format (1
op *`_` : Exp -> Exp [strict prec 25] .
op !`_` : Exp -> Exp [aux] .
vars E E' : Exp .
eq ! E = E ? 0 : 1 .
op _&&_ : Exp Exp -> Exp [aux latex(renameTo _\&\&_)] .
op _||_ : Exp Exp -> Exp [aux latex(renameTo _\ensuremath`{`|`|`}_)] .
eq E && E' = E ? E' : 0 .
eq E || E' = E ? 1 : E' .
op _?`_` : Exp Exp Exp -> Exp [format (d b o b o d) renameTo if`(_)`_else_ prec 39] .
op _:=_ : Exp Exp -> Exp [strict(2) prec 40 gather (e E)] .
op _; : Exp -> Stmt [prec 45 strict format (d b o)] .
op ; : -> Stmt [renameTo .K] .
op _ : StmtList StmtList -> StmtList [prec 100 gather(e E) format (d ni d) renameTo _->_] .
op `{`_`} : StmtList -> Stmt [format (b on+i -nbi o) renameTo _ latex(renameTo _\`{`_`})] .
op `{`_`} : -> Stmt [format (b d o) renameTo .K latex(renameTo _\`{`_`})] .
op malloc`(_)` : Exp -> Exp [format (g b o b o) strict] .
op free`(_)` : Exp -> Exp [format (g b o b o) strict] .
op if`(_)`_ : Exp Stmt -> Stmt [aux prec 47] .
var St St' : Stmt .
```

```

eq if(E) St = if (E) St else {} .
op if`(_`)_else_ : Exp Stmt Stmt -> Stmt [strict (1) format(g b o b os nig o d) prec 46] .
op while`(_`)_ : Exp Stmt -> Stmt [syntax format (g b o b os d)] .
op printf`("%d ",_) : Exp -> Exp [format (g b r b o b o) strict latex(renameTo printf\mybracket`\my
op null : -> Exp [aux] .
eq null = 0 .
k)

```

Note that the syntax module is introduced by

```

(k syntax for KERNELC is
...
k)

```

Beside the standard Maude operation attributes (prec, gather, format, prec), notice the K-specific ones:

- **strict** and **strict(n1 n2 ... nk)** specify what arguments need to be evaluated before evaluating the termed topped in that language construct. For example, in the following definition,

```

op _:=_ : Exp Exp -> Exp [strict(2) prec 40 gather (e E)] .

```

**strict(2)** says that the semantic rule for assignment will assume the second argument is evaluated. Note that the evaluation of the left-hand-side to a L-value is only a partial evaluation, so it cannot be specified as a strictness attribute here. #Semantics section below describes one way to specify it.

- **renameTo** is used to rename an operation into another one, either to avoid parsing problems, or to desugar some constructs into other K constructs. For example, in the definition

```

op _?_:_ : Exp Exp Exp -> Exp [format (d b o b o d) renameTo if`(_`)_else_ prec 39] .

```

**renameTo if`(\_`)\_else\_** specifies that **\_?\_:\_** would be interpreted as **if`(\_`)\_else\_** for semantics purposes. Similarly, in the definition

```

op `{_`_ : StmtList -> Stmt [format (b on+i -nbi o) renameTo _] .

```

**renameTo \_** specifies that **{\_`\_}** is no longer needed once the parsing is done, since it was there only for grouping reasons.

- **aux** is used to denote that a certain operation is completely desugared (through equations) in this syntax module and will not appear in the semantics definition. For example, in the definition,

```

op if`(_`)_ : Exp Stmt -> Stmt [aux prec 47] .

```

**aux** is used to say that, although we accept conditionals without the else branch, they would not be handled specifically in the semantics, since they are desugared by the next equation:

```

eq if(E) St = if (E) St else {} .

```

**latex** attribute is used to wrap attributes altering the latex generator. in this context **renameTo** says how the operator should be displayed in latex

## Configuration

```
(k configuration for KERNELC is
including KMAP{K, K} + FRESH-ITEM{K} .
ops env mem ptr : -> CellLabel [wrapping Map`{K`,K`}]] .
op out : -> CellLabel [wrapping K] .
op stream : String -> K .
op void : -> KResult .
k)
```

The configuration module is introduced by

```
(k configuration for KERNELC is
...
k)
```

It specifies what Labels will be used to wrap configuration items, and might declare additional helping operations to be used in the semantics. All declared labels must be constants of sort CellLabel and must specify what sort of items they would wrap by using the *wrapping* attribute. Declaring a label, say **env**, it allows for a tag-like operation **<env>** **</env>** to be used in the semantics definition. The **k** CellLabel wrapping computations (of sort **K**) and the **T** CellLabel, wrapping a soup of configuration items are predefined. This tag-like wrapper corresponds to the cell structure used in mathematical notation of K; for example **<k> K </k>** corresponds to  $\langle K \rangle_k$ . Open ended K cells are written in the tag notation as follows:

- $\langle k \rangle_k$  is written as **<k> K ...</k>**
- $\langle Env \rangle_{env}$  is written as **<env>... Env ...</env>**
- $\langle Str \rangle_{out}$  is written as **<out>... Str </out>**

## Semantics

```
(k semantics for KERNELC is
including GENERIC-EXP-SEMANTICS .
var P : Pgm . var N N' : Nat . var X : Name .
var Env : Map{K,K} . var V V' : KResult .
var I : Int . var Ptr Mem : Map{K,K} .
var K K1 K2 : K . var S : String .
--- evaluating the lhs of an equality to LVal
kcxt * K1 := K2 [strict(K1)] .
op run : Pgm -> Config .
eq run(P)
= <T>
  <k> mkK(P) </k> <env> .empty </env>
  <mem> .empty </mem> <ptr> .empty </ptr>
  <nextItem> item(1) </nextItem>
  <out> stream("") </out>
</T> .
eq #(true) = #(1) .
eq #(false) = #(0) .
eq if (#(0)) _ else K2 = K2 .
ceq if (#(I)) K1 else _ = K1 if I neq 0 .
--- discarding the value of an expression statement .
eq V ; = .K .
keq <k> [ [X ==> V]] ...</k>
  <env>... X |-> V ...</env> .
keq <k> [ [X := V ==> V]] ...</k>
  <env> [ [Env ==> Env[X <- V]]] </env> .
keq <k> [ [* #(N) ==> V]] ...</k>
  <mem>... #(N) |-> V ...</mem> .
```

```

keq <k> [ [* #(N) := V ==> V]] ...</k>
  <mem>... #(N) |-> [[_ ==> V]] ...</mem> .
keq <k> [ [while (K1) K2 ==> if (K1) (K2 -> while(K1) K2) else .K]] ...</k> .
op alloc : Nat Nat -> Map{K,K} .
eq alloc(N, 0) = .empty .
eq alloc(N, s(N')) = (#(N) |-> #(0)) &' alloc(N + 1, N') .
op freeMem : Map{K,K} Nat Nat -> Map{K,K} .
eq freeMem(Mem, N, 0) = Mem .
eq freeMem((Mem &' (#(N) |-> V)), N, s(N')) = freeMem(Mem,N + 1,N') .
keq <k> [ [ malloc(#(N)) ==> #(N')] ] ...</k>
  <ptr>... [ [.empty ==> (#(N') |-> #(N))] ] ...</ptr>
  <nextItem> [ [item(N') ==> item(N') + N]] </nextItem>
  <mem>... [ [.empty ==> alloc(N', N)] ] ...</mem> .
keq <k> [ [free(#(N)) ==> void]] ...</k>
  <ptr>... [ [#(N) |-> #(N') ==> .empty]] ...</ptr>
  <mem> [ [Mem ==> freeMem(Mem, N, N')] ] </mem> .
keq <k> [ [printf("%d ",#(I)) ==> void]] ...</k>
  <out> [ [stream(S) ==> stream(S + string(I,10)+ " ")] ] </out> .

```

The semantics module is introduced by

```

(k semantics for KERNELC is
...

```

K-specific constructs which can be found in this module are K-contexts, K-equations, and K-rules, each being introduced by **kcxt**, **keq**, and **krl**, respectively. Moreover, K-equations and K-rules, provide notational shortcuts to make the equations/definitions mode compact.

- K-Contexts specify strictness constraints, and are usually used for those constraints which depend on a context rather than a single construct. For example,

```

kcxt * K1 := K2 [strict(K1)] .

```

specifies the evaluation to a L-value of a pointer in the assignment construct, by saying that when giving the semantics to that context, K1 would need to be a value.

- K-equations and K-rules have the same syntax, except the first would be desugared into equations, while the latter into rules. Basically, a K-equation/rule is a term which should contain at least one occurrence of the  $[[T1 ==> T2]]$  construct, which is used as a textual representation of the K visual replacement pattern:  $\frac{T1}{T2}$ .

For K equations and K rules, one can use ``_`` (that is, underscore) to specify anonymous variables.

## Putting them all together

The user needs to load, in order the modules for syntax, configuration, and semantics. Since these modules are internally desugared into Maude modules, they would receive unique ids. For our KERNELC example, these are as follows:

- syntax generates two modules: KERNELC-SYNTAX which only contains the syntax, as defined by the user, and KERNELC-K-SYNTAX which additionally contains the K abstract syntax associated to it and a syntax transformer mkK from the user syntax to the internal K syntax, as well as equations meant to satisfy the strictness constraints.
- configuration generates one module KERNELC-K-COFIGURATION, similar to the one introduced by the user

and,

- semantics generates the module KERNELC-K-SEMANTICS which in big lines follows the semantic definition introduced by the user, but desugars the k-contexts, k-equations, and k-rules, and transforms everything into the K abstract syntax so that the definition becomes executable.

Therefore, the thus defined K modules can be referred to from usual Maude modules, once the K-modules have been processed. For example, a module defining KERNELC programs can be introduced right after the k syntax definition, and it can refer to it:

```
(mod KERNELC-PROGRAMS is including KERNELC-SYNTAX .
op pSum : -> Pgm .
eq pSum =
  #include<stdio.h>
  #include<stdlib.h>
  void main(void) {
    n := 1000 ;
    s := 0 ;
    i := n ;
    while (!(i <= 0)) {
      s := s + i ;
      i := i - 1 ;
    }
    printf("%d ",s);
  }
.
endm)
```

Likewise, the following module can be used to run programs once the k semantics module has been introduced:

```
(mod TEST is
including KERNELC-K-SEMANTICS .
including KERNELC-PROGRAMS .
endm)
```

The following command executes the pSum program defined above.

```
(red < T > pSum </ T > .)
```

Note that, outside of the K definitions, the tags need to be properly separated with spaces.

## Entire definition in one module, and a more complex language -- KCHALLENGE

```
(k definition for KCHALLENGE is
including INT-K-SYNTAX + BOOL-K-SYNTAX + FLOAT-K-SYNTAX .
including NAME-K-SYNTAX .
including COMMON-ENV-STORE .
including BINDER-K-SYNTAX .
including KMAP{K,K} + KSET{K} .
including TUPLE .
```

```
var VL : List{KResult} . var N : Nat .
vars I I1 I2 : Int . var B : Bool . var X : Name . var V V' : KResult .
vars F1 F2 : Float . var L L' : Loc .
var E BE S S1 S2 K K' : K . var Sigma : Store . var Rho Rho' : Env{Loc} .
```

```
xsort Exp .
```

*xsort* is used to specify a syntax sort -- all operations having it as a result sort would be considered part of syntax

```
subsort #Int #Bool #Float < Exp .
```

By convention, # sorts are used for input literals, while their corresponding semantics sorts are named by removing the #. Int, Bool, and Floa, are imported from built-in modules INT-K-SYNTAX, BOOL-K-SYNTAX, and FLOAT-K-SYNTAX, respectively.

```
subsort #Name < Exp .  
krl <k> [[X ==> V]] ...</k> <env>... X |-> L ...</env> <store>... L |-> V ...</store> .
```

Names are imported from predefined module NAME-K-SYNTAX. *env* and *store* cells as well as basic operations on them are imported from COMMON-ENV-STORE. When a name is matched at the top of computation, it can be replaced by value V, provided that the mapping of X to location L can be matched in the environment and the mapping of L to V can be matched in the store.

```
op _+_ : Exp Exp -> Exp [gather(E e) prec 33 strict] .  
rl # I1 + # I2 => # (I1 + I2) .  
rl # F1 + # F2 => # (F1 + F2) .
```

*gather* and *prec* are standard attributes in Maude, used for parsing. Additionally *strict* specifies that both operands of + should be evaluated (order not specified) before + being evaluated itself. This allows us to only give semantics of + on valued terms. operation # is used to distinguish values when giving semantics.

```
op *__ : Exp Exp -> Exp [gather(E e) prec 31 strict] .  
rl *__(# I1, # I2) => # (I1 * I2) .  
rl *__(# F1, # F2) => # (F1 * F2) .
```

```
op _<=_ : Exp Exp -> Exp [prec 37 seqstrict] .  
rl # I1 <= # I2 => # (I1 <= I2) .  
rl # F1 <= # F2 => # (F1 <= F2) .
```

*seqstrict* is specified when the order of evaluation matters. The default order is left-to-right.

```
op not_ : Exp -> Exp [prec 53 strict] .  
rl not (# B) => # (not B) .
```

```
op _and_ : Exp Exp -> Exp [gather(E e) prec 55 strict(1)] .  
rl (# true) and BE => BE .  
rl (# false) and _ => # false .
```

The semantics of *and* given above is '*shortcut*'. For this, we only require the first operand to be evaluated -- *strict(1)* -, and evaluate the expression according to that value. Note that one can use anonymous variable \_ if the name of a variable is only required for matching.

```
xsort Stmt .  
op _;_ : Stmt Stmt -> Stmt [prec 100 gather(e E) renameTo _->_] .
```

By renaming the statement sequencing operator to  $\_ \rightarrow \_$ , the K sequencing operator, we don't need to specify any additional semantic rules for it.

```
op var_ : #Name -> Stmt .
krl <k> [[var X ==> .K]] ...</k>
    <env> [[Rho ==> Rho[X <- L]]] </env>
    <nextLoc> [[L ==> L + 1]] </nextLoc> .
```

Variable declaration: we use the *nextLoc* cell imported from COMMON-ENV-STORE. also, we use  $\text{Rho}[X \leftarrow L]$  to map  $X$  to  $L$  in  $\text{Rho}$ .

```
op if_then_else_ : Exp Stmt Stmt -> Stmt [strict(1)] .
rl if (# true) then S1 else _ => S1 .
rl if (# false) then _ else S2 => S2 .
```

To give semantics for conditional, we only need to evaluate the condition-- *strict(1)*

```
op while_do_ : Exp Stmt -> Stmt .
keq <k> [[while BE do S ==> if BE then (S -> while BE do S) else .K]] ...</k> .
```

The semantics of *while* is given by unrolling, but only at the top of the computation, to avoid cycling.

```
op _;_ : Stmt Exp -> Exp [prec 110 renameTo _->_] .

op output_ : Exp -> Stmt [strict] .
op output : -> CellLabel [wrapping List`{KResult`}]] .
krl <k> [[output V ==> .K]] ...</k> <output>... [[.nil ==> V]] </output> .
```

We declare a new cell to hold the list of values.

```
op ++_ : #Name -> Exp [prec 0 renameTo inc] .
krl <k> [[inc(X) ==> # (I + 1)]] ...</k> <env>... X |-> L ...</env>
    <store>... L |-> # [[I ==> I + 1]] ...</store> .
```

Sometime we need to rename constructs when going from syntax to semantics to avoid name clashing. In this case, we rename *++* to *inc*.

```
op #_ : Loc -> KResult .
op &_ : #Name -> Exp [prec 0] .
krl <k> [[& X ==> # L]] ...</k>
    <env>... X |-> L ...</env> .
```

Obtaining the reference location of a name.

```
op *_ : Exp -> Exp [strict prec 25] .
krl <k> * # L ==> V? ...</k> <store>... L |-> V ...</store> .
```

Dereferencing a location.

```
op ref_ : Exp -> Exp [strict prec 0] .
krl <k> [[ref V ==> L]] ...</k>
    <store> [[Sigma ==> Sigma[L <- V]]] </store>
    <nextLoc> [[L ==> L + 1]] </nextLoc> .
```



Evaluating an expression and returning a reference to it.

```
op _:=_ : Exp Exp -> Stmt [strict(2)] .
krl <k> [[X := V ==> .K]] ...</k>
    <env>... X |-> L ...</env>
    <store> [[Sigma ==> Sigma[L <- V]]] </store> .
kcxt * K := K' [strict(K)] .
krl <k> * # L := V ==> .K? ...</k>
    <store> [[Sigma ==> Sigma[L <- V]]] </store> .
```

Assignment is similar to C/KernelC -- the right hand side is evaluated to a value, specified here by *strict(2)*, while the left-hand-side is evaluated to a l-value, here either a Name, or the dereferencing of a location.

```
op aspect_ : Stmt -> Stmt .
op aspect : -> CellLabel [wrapping K] .
krl <k> [[aspect S ==> .K]] ...</k> <aspect> [[_ ==> S]] </aspect> .
keq <k> [[lambda X . E ==> closure(X, S -> E, Rho)]] ...</k>
    <env> Rho </env> <aspect> S </aspect> .
```

Aspects are defined to be executed before the start of each function evaluation, in the environment in which that function was defined. Note that lambda abstractions are evaluated to closures.

```
op __ : Exp Exp -> Exp [strict renameTo apply] .
op closure : Name K Env{Loc} -> KResult .
krl <k> [[ apply(closure(X, E, Rho), V) ==> E -> restore(Rho')]] ...</k>
    <env> [[Rho' ==> Rho[X <- L]]] </env>
    <store> [[Sigma ==> Sigma[L <- V]]] </store>
    <nextLoc> [[L ==> L + 1]] </nextLoc> .
```

Application of a function is declared *strict* (call-by-value), renamed to *apply* to avoid name clashes. Note that we use *restore* (imported from COMMON-ENV-STORE) to restore the environment once the function was evaluated.

```
krl <k> [[ mu X . E ==> E -> restore(Rho)]] ...</k>
    <env> [[Rho ==> Rho[X <- L]]] </env>
    <store> [[Sigma ==> Sigma[L <- mu X . E]]] </store>
    <nextLoc> [[L ==> L + 1]] </nextLoc> .
```

The semantics of  $\mu X.E$  is given by evaluated the expression in the environment/store where **X** is bound to a  $\mu X.E$ .

```
op callcc_ : Exp -> Exp [strict] .
op cc : K Env{Loc} -> KResult .
krl <k> [[callcc V ==> apply(V, cc(K, Rho))]] -> K </k>
    <env> Rho </env> .
krl <k> [[apply(cc(K, Rho), V) -> _ ==> V -> K]] </k>
    <env> [[_ ==> Rho]] </env> .
```

```
op randomBool : -> Exp .
knd <k> [[randomBool ==> # true]] ...</k> .
knd <k> [[randomBool ==> # false]] ...</k> .
```

*randomBool* non-deterministically evaluates to either **true** or **false**.

```
op spawn_ : Stmt -> Stmt .
op holds : -> CellLabel [wrapping Map`{K`,K`}]] .
```

```

op thread : -> CellLabel [wrapping Set`{ConfigItem`}]] .
var Holds : Map`{K`,K`} .
krl <thread>... <k> [[spawn S ==> .K]] ...</k> <env> Rho </env> ...</thread>
    [[.empty ==> .empty <thread> <k> S </k> <env> Rho </env>
        <holds> .empty </holds> </thread>]] .

```

Spawning a new thread. *thread* cell is used to group together all cell related to a thread; *holds* holds the locks acquired by the thread (and their multiplicity).

```

op busy : -> CellLabel [wrapping Set`{K`}]] .
var Busy : Set`{K`} .
krl [[<thread>... <k> .K </k> <holds> Holds </holds> ...</thread>
    ==> .empty]] <busy> [[Busy ==> Busy -' keys(Holds)]] </busy> .

```

When the computation of a thread has completed, it can be dissolved and its resources released. *busy* holds the name of the locks acquired by any of the threads.

```

op acquire_ : Exp -> Stmt [strict] .
krl <k>[[acquire V ==> .K]] ...</k>
    <holds>... V |-> # [[N ==> s(N)]] ...</holds> .
kcrl <k>[[acquire V ==> .K]] ...</k>
    <holds>... [[.empty ==> V |-> # 0]] ...</holds>
    <busy> Busy [[.empty ==> V]]</busy> if not(V in' Busy) .

```

One can lock on any value. A lock can only be acquired if it is not busy. Same thread can acquire same lock multiple times, therefore we keep multiplicities for each lock in the *holds* cell.

```

op release_ : Exp -> Stmt [strict] .
krl <k>[[release V ==> .K]] ...</k>
    <holds>... V |-> # [[s(N) ==> N]] ...</holds> .
krl <k>[[release V ==> .K]] ...</k>
    <holds>... -> # 0 ==> .empty? ...</holds>
    <busy>... [[V ==> .empty]] ...</busy> .

```

```

op rv_ : Exp -> Stmt [strict] .
krl <k> [[rv V ==> .K]] ...</k> <k> [[rv V ==> .K]] ...</k> .

```

Rendez-vous synchronization. Two threads can only pass together a barrier specified by a lock V.

```

sort Agent .
op agent : Nat -> Agent .
subsort Agent < KResult .
op new-agent_ : Stmt -> Exp .
op agent : -> CellLabel [wrapping Set`{ConfigItem`}]] .
ops me parent nextAgent : -> CellLabel [wrapping Agent] .
krl <k> [[new-agent S ==> agent(N)]] ...</k> <me> Me </me>
    <nextAgent> agent([[N ==> N + 1]]) </nextAgent>
    [[.empty ==> <agent>
        <thread>
            <k> S </k> <env> .empty </env> <holds> .empty </holds>
        </thread>
        <store> .empty </store> <nextLoc> loc(0) </nextLoc>
        <aspect> .K </aspect> <busy> .empty </busy>
        <me> agent(N) </me> <parent> Me </parent>
    </agent>]] .

```

An agent is here a collection of threads, grouped in an *agent* cell identified by an id held by the *me* cell, and holding in the *parent* cell a reference id to its creating agent. *nextAgent* is used for providing fresh ids for agents.

```

rl <agent>
  <store> _ </store>
  <nextLoc> _ </nextLoc>
  <aspect> _ </aspect>
  <busy> _ </busy>
  <me> _ </me>
  <parent> _ </parent>
</agent> ==> .empty .

```

When all threads inside an agent have completed, the agent can be dissolved.

```

op me : -> Exp .
krl <k>[[me ==> A]] ...</k> <me> A </me> .

```

```

op parent : -> Exp .
krl <k>[[parent ==> A]] ...</k> <parent> A </parent> .

```

```

op message : -> CellLabel [wrapping Tuple] .
vars Me Parent A : Agent .
op send-async__ : Exp Exp -> Stmt [strict] .
krl <k> [[send-async A V ==> .K]] ...</k> <me> Me </me>
  [[<message> [Me,A,V] </message> ==> .empty]] .

```

An agent can send any value (including agents ids) to other agents (provided it knows their id). To model asynchronous communication, each value sent is wrapped in a *message* cell identifying both the sender and the intended receiver.

```

op receive-from_ : Exp -> Exp [strict] .
krl <k> [[receive-from A ==> V]] ...</k> <me> Me </me>
  [[<message> [A,Me,V] </message> ==> .empty]] .

```

An agent can request to receive a message from a certain agent.

```

ops receive : -> Exp .
krl <k> [[receive ==> V]] ...</k> <me> Me </me>
  [[<message> [_ ,Me,V] </message> ==> .empty]] .

```

An agent can request to receive a message from any agent.

```

op send-synch__ : Exp Exp -> Stmt [strict] .
krl <agent>... <k> [[send-synch A V ==> .K]] ...</k> <me> Me </me> ...</agent>
  <agent>... <k> [[receive-from Me ==> V]] ...</k> <me> A </me> ...</agent> .
krl <agent>... <k> [[send-synch A V ==> .K]] ...</k> ...</agent>
  <agent>... <k> [[receive ==> V]] ...</k> <me> A </me> ...</agent> .

```

The message can be sent synchronously, in which case, two agents need to be matched together for the exchange to occur.

```

op halt : -> Stmt .
krl [[<agent>... <k> halt ...</k> ...</agent>
==> .empty]] .

```

The semantics of halt in one of the thread of an agent is that it dissolves the agent.

```
op messages : -> CellLabel [wrapping Set`{ConfigItem`}] .
op result : -> CellLabel [wrapping List`{KResult`}] .
kconf
  <T>
    <agent*>
      <thread*>
        <k> _ </k>
        <env> _ </env>
        <holds> _ </holds>
      </thread*>
      <store> _ </store>
      <nextLoc> _ </nextLoc>
      <aspect> _ </aspect>
      <busy> _ </busy>
      <me> _ </me>
      <parent> _ </parent>
    </agent*>
    <output> _ </output>
    <messages> <message*> _ </message*> </messages>
    <nextAgent> _ </nextAgent>
  </T> <result> _ </result> .
```

For K to know where each cell is located, one needs to specify (if there are cells at different levels) the structure of the configuration, together with an indication of which of the cells have multiplicities. Notice here that we have a wrapper for messages which was not specified anywhere, as well as a wrapper for the results which is at the top level.

```
op `[_[_]` : Stmt -> Config .
eq P:Stmt?
  = <T>
    <agent>
      <thread>
        <k> mkK(P:Stmt) </k>
        <env> .empty </env>
        <holds> .empty </holds>
      </thread>
      <store> .empty </store>
      <nextLoc> loc(0) </nextLoc>
      <aspect> .K </aspect>
      <busy> .empty </busy>
      <me> agent(0) </me>
      <parent> agent(0) </parent>
    </agent>
    <output> .nil </output>
    <messages> .empty </messages>
    <nextAgent> agent(1) </nextAgent>
  </T> .
```

This is how a program is initialized to be executed using the above definition.

```
krl [[<T> <output> VL </output> <nextAgent> _ </nextAgent> <messages> _ </messages> </T>
==> <result> VL </result> ]] .
```

When there are no more agents executing, we can collect the output and transfer it into the result cell.

Advanced features: program generation

```
op quote_ : Exp -> Exp .
```

```

op unquote_ : Exp -> Exp .
op eval_ : Exp -> Exp [strict] .

op quote : Nat List{K} -> KProper .
op code : List{K} -> KResult .
op _box`(->`)_ : K K -> KProper [strict] .
op _box`(`,`)_ : K K -> KProper [strict] .

op box : K -> KSynLabel .
op kl : KLabel -> K .
var KL : KLabel . var KKL : K . var Ks : NeList`{K`} .
var K1 K2 : NeK .
kcxt box(KKL)(Ks) [strict(Ks)] .

```

```

keq <k> [[quote K ==> quote(0,K)]] ...</k> .
eq quote(N, K1 -> K2) = quote(N, K1) box(->) quote(N,K2) .
eq code(K1) box(->) code(K2) = code(K1 -> K2) .
ceq quote(N, KL(Ks)) = box(kl(KL))(quote(N, Ks))
  if KL /= quote~ /\ KL /= unquote~ .
eq box(kl(KL))(code(Ks)) = code(KL(Ks)) .
eq quote(N, quote(K)) = box(kl(quote~))(quote(s(N), K)) .
eq quote(0, unquote(K)) = K .
eq quote(s(N), unquote(K)) = box(kl(unquote~))(quote(N,K)) .

```

```

eq quote(N, (K, Ks)) = quote(N, K) box(,) quote(N, Ks) .
eq code(K) box(,) code(Ks) = code((K, Ks)) .

```

```

eq quote(N, V) = code(V) .
eq quote(N,X) = code(X) .

```

```

eq eval code(K) = K .

```

k)

As for the previous definitions, we end our specification with *k*).

## Using Core Maude

Writing K definitions in Core Maude is similar to the process described above, but, since the definitions must be recognizable as Maude modules, several artifices, which will be described below, are used to encode them into standard Maude syntax.

Using Core Maude has the advantage that the process of conversion of the input modules needs to only be performed when the input module changes, and therefore the loading time of the thus pre-compiled modules is much shorter (especially since Full-Maude does not need to be loaded once the generation process is completed). Moreover it is less likely that the compilation process will crash Maude in case of errors, as it might happen when using the Full-Maude version (we assume the guilt here — it is not because of Full-Maude, but rather because of our messing with its internal structures).

The drawback is that the user has to go back and forth by introducing a module, compiling it to its executable Maude correspondent(s), writing the result of the compilation into a file; and this process needs to be repeated for each of the modules.

Scenario for using it K-Maude in Full-Maude:

```

load <base-dir>/k-prelude

```

## 1. Write the syntax module KERNELC-SYNTAX

[Optional] write some programs using it, to check the definition is correct

2. compile the definition to obtain module KERNELC-K-SYNTAX, and save the generated module in some file
3. write the KERNELC-CONFIGURATION module,
4. compile the definition to obtain module KERNELC-K-CONFIGURATION, and save the generated module in some file
5. write the KERNELC-SEMANTICS module,
6. compile the definition to obtain module KERNELC-K-SEMANTICS, and save the generated module in some file

[Optional] load KERNELC-K-SEMANTICS and use it to execute programs

Let us exemplify the above steps through the corresponding K-Core-Maude definition for KERNELC. For each of the modules we will present the non-compiled version, and discuss how K-specific constructs are encoded in the definitions, as well as the compilation process.

### Syntax

```
mod KERNELC-SYNTAX is including GENERIC-EXP-K-SYNTAX + STRING-K-SYNTAX .
sorts Stmt StmtList Pgm .
subsort Stmt < StmtList .
op #include<stdio.h>`#include<stdlib.h>`void`main`(void`){_`} : StmtList -> Pgm [metadata "renameTo _"] .
op *_ : Exp -> Exp [metadata "strict" prec 25] .
op !_ : Exp -> Exp [metadata "aux"] .
vars E E' : Exp .
eq ! E = E ? 0 : 1 .
ops _&&_ _||_ : Exp Exp -> Exp [metadata "aux"] .
eq E && E' = E ? E' : 0 .
eq E || E' = E ? 1 : E' .
op _?_ : Exp Exp Exp -> Exp [format (d b o b o d) metadata "renameTo if`(_)`_else_" prec 39] .
op _= : Exp Exp -> Exp [metadata "strict(2) renameTo _:=_" prec 40 gather (e E)] .
op _; : Exp -> Stmt [prec 45 metadata "strict" format (d b o)] .
op ; : -> Stmt [metadata "renameTo .K"] .
op __ : StmtList StmtList -> StmtList [prec 100 gather(e E) format (d ni d) metadata "renameTo _->_"] .
op {_} : StmtList -> Stmt [format (b on+i -nbi o) metadata "renameTo _"] .
op {} : -> Stmt [format (b d o) metadata "renameTo .K"] .
op malloc`(_)` : Exp -> Exp [format (g b o b o) metadata "strict"] .
op free`(_)` : Exp -> Exp [format (g b o b o) metadata "strict"] .
op if`(_)`_ : Exp Stmt -> Stmt [metadata "aux" prec 47] .
var St St' : Stmt .
eq if(E) St = if (E) St else {} .
op if`(_)`_else_ : Exp Stmt Stmt -> Stmt [metadata "strict (1)" format(g b o b os nig o d) prec 46] .
op while`(_)`_ : Exp Stmt -> Stmt [format (g b o b os d) metadata "syntax"] .
op printf`("%d",_)` : Exp -> Exp [format (g b r b o b o) metadata "strict"] .
op NULL : -> Exp [metadata "aux"] .
eq NULL = 0 .
endm
```

Note that all K-specific attributes have been encoded into the metadata attribute.

To compile the above module, one would need to execute the following commands:

```
load <base-dir>/make-k
(makeKSyntax KERNELC .)
```

The output of the last command will be a module named KERNELC-K-SYNTAX containing the abstract syntax, and the strictness equations.

## Configuration

```
mod KERNELC-CONFIGURATION is including KERNELC-K-SYNTAX + KMAP{K, K} + FRESH-ITEM{K} + CONFIG .
ops env mem ptr : -> CellLabel [metadata "wrapping Map{K,K}"] .
op out : -> CellLabel [metadata "wrapping K"] .
op stream : String -> K .
op void : -> KResult .
endm
```

Again, operation attributes are encoded in the metadata attribute. Also note that the generated syntax module (KERNELC-K-SYNTAX) and the k-prelude CONFIG module must be now included explicitly (they needed not be mentioned in the Full-Maude version).

To compile the above module, one would need to execute the following commands:

```
load <base-dir>/make-k
(makeKConfiguration KERNELC .)
```

The output of the last command will be a module named KERNELC-K-CONFIGURATION containing the tag-like operations which would be used in the semantics definition.

## Semantics

```
mod KERNELC-SEMANTICS is including KERNELC-K-CONFIGURATION + GENERIC-EXP-SEMANTICS + K-RULES + INT-SIMP
var P : Pgm .
var N N' : Nat . var X : Name .
var Env : Map{K,K} . var V V' : KResult .
var I : Int . var Ptr Mem : Map{K,K} .
var K K1 K2 : K . var S : String .
--- evaluating the lhs of an equality to LVal
mb kcxt * K1 := K2 : K [metadata "strict(K1)"] .
op [ [_] ] : Pgm -> Config .
eq [ [ P ] ]
= <T>
  <k> mkK(P) </k> <env> .empty </env>
  <mem> .empty </mem> <ptr> .empty </ptr>
  <nextItem> item(1) </nextItem>
  <out> stream("") </out>
</T> .
eq #(true) = #(1) .
eq #(false) = #(0) .
ceq if (#(I)) K1 else K2 = K2 if I eq 0 .
ceq if (#(I)) K1 else K2 = K1 if I neq 0 .
--- discarding the value of an expression statement .
eq V ; = .K .
mb keq <k> [ [X ==> V] ] ...</k>
  <env>... X |-> V ...</env> : K .
mb keq <k> [ [X := V ==> V] ] ...</k>
  <env> [ [Env ==> Env[X <- V]] ] </env> : K .
mb keq <k> [ [* #(N) ==> V] ] ...</k>
  <mem>... #(N) |-> V ...</mem> : K .
mb keq <k> [ [* #(N) := V ==> V] ] ...</k>
  <mem>... #(N) |-> [ [V' ==> V] ] ...</mem> : K .
mb keq <k> [ [while (K1) K2 ==> if (K1) (K2 -> while(K1) K2) else .K] ] ...</k> : K .
op alloc : Nat Nat -> Map{K,K} .
eq alloc(N, 0) = .empty .
eq alloc(N, s(N')) = (#(N) |-> #(0)) &' alloc(N + 1, N') .
```

```

op freeMem : Map{K,K} Nat Nat -> Map{K,K} .
eq freeMem(Mem, N, 0) = Mem .
eq freeMem((Mem & ' (#(N) |-> V)), N, s(N')) = freeMem(Mem,N + 1,N') .
mb keq <k> [ [ malloc( #(N) ) ==> #(N') ] ] ...</k>
      <ptr>... [ [.empty ==> ( #(N') |-> #(N) ) ] ] ...</ptr>
      <nextItem> [ [ item(N') ==> item(N') + N ] ] </nextItem>
      <mem>... [ [.empty ==> alloc(N', N) ] ] ...</mem> : K .
mb keq <k> [ [ free( #(N) ) ==> void ] ] ...</k>
      <ptr>... [ [ #(N) |-> #(N') ==> .empty ] ] ...</ptr>
      <mem> [ [ Mem ==> freeMem(Mem, N, N') ] ] </mem> : K .
mb keq <k> [ [ printf("%d ", #(I) ) ==> void ] ] ...</k>
      <out> [ [ stream(S) ==> stream(S + string(I,10)+ " ") ] ] </out> : K .
endm

```

Note that the generated configuration module (KERNELC-K-CONFIGURATION) and the k-prelude K-RULES module must be now included explicitly (they needed not be mentioned in the Full-Maude version). K-contexts, K-equations, and K-rules are encoded as membership axioms to sorts Kcxt, Keq, and Krl, respectively. Note that membership axioms are only used as a notation; there is no membership-related semantics associated to them. For example, the K-equation for retrieving a value form the environment

```

keq <k> [ [ X ==> V ] ] ...</k>
    <env>... X |-> V ...</env> .

```

would be written using this convention as

```

mb keq <k> [ [ X ==> V ] ] ...</k>
      <env>... X |-> V ...</env> : K .

```

Similarly, the K-context definition

```

kcxt * K1 := K2 [strict(1)] .

```

would be written as

```

mb kcxt * K1 := K2 : K [metadata "strict(K1)"] .

```

using metadata to encode the strictness attribute.

To compile the above module, one would need to execute the following commands:

```

load <base-dir>/make-k
(makeKSemantics KERNELC .)

```

The output of the last command will be a module named KERNELC-K-SEMANTICS containing the executable semantic definition.

## Scripts automating the process

Several **bash** scripts (should work on unix/linux, might also work on windows under cygwin) are provided in the base directory of K, to automate the generation process for the Core Maude version. These are **makeK** which takes a file containing one of the syntax/configuration/semantics modules and generates the corresponding K module, **buildK**



which runs **makeK** to generate all files at once (good for already written definitions), and **cleanK** which removes the files generated by the other scripts. This script only work assuming certain conventions are followed in naming and structuring the files containing the modules (which are followed by all the included examples). These conventions are:

- **makeK** and **buildK** should not be moved from the base directory of K (or executed through links), since they use their location to discover that base directory;
- assuming the name of the language is **LANG**, the name of the syntax/configuration/semantics modules written by the user should be **LANG-SYNTAX**, **LANG-CONFIGURATION**, and **LANG-SEMANTICS**, and each should be placed in one file named **lang-syntax.maude**, **lang-configuration.maude**, and **lang-semantics.maude**, respectively; the generated modules would then be **LANG-K-SYNTAX**, **LANG-K-CONFIGURATION**, and **LANG-K-SEMANTICS**, and each would be placed in one file named **lang-k-syntax.maude**, **lang-k-configuration.maude**, and **lang-k-semantics.maude**, respectively;
- **makeK** takes as argument the name of the file containing the module which should be *compiled* (with or without extension), and should generate the corresponding K module and write it in the corresponding file;

**Important:** the file should be self contained and loadable in maude without significant errors, i.e., it should specify inside the files it depends on (such as **k-prelude**, **lang-k-syntax**, **lang-k-configuration**, and so on); that is why **makeK** should be run in order on the files to be generated.

**Important:** the name of the Maude executable (including path, if necessary) should be set on line 2 of **makeK**

- **buildK** takes as argument *lang* runs **makeK** on the three *lang* files, in order; it should therefore only be run if those three files reside in the same directory;
- **cleanK** takes as argument *lang* and removes all files in the directory from which they are run whose name start with *lang-k-*; if you are not following conventions, better not use it since it might remove files you don't want removed.

## Generating Latex

Currently this is less stable than the tool itself, and works only for full Maude definitions. To generate LaTeX one can use the command (*print definition KERNELC .*) after loading the semantics file, and then copy the output to a tex file. Alternatively, script **compileL** attempts to automate this process, taking two arguments: the name of the file (e.g., *kernelc-semantics*) and the name of the definition (e.g., *KERNELC*), and producing a tex file containing the generated LaTeX, named by appending extension *.tex* to the provided file name. the scripts also attempts to compile it using pdflatex.

Adus de la "[http://fsl.cs.uiuc.edu/index.php/K\\_Prototype](http://fsl.cs.uiuc.edu/index.php/K_Prototype)"

---



- Ultima modificare 23:38, 27 octombrie 2009.